

GRAPHICAL MODEL PREPARATION FOR EMBEDDED DEPLOYMENT

Field of the Invention

The present invention relates to the generation of code from a graphical model, such as a block diagram. More particular, the present invention relates to a process of preparing a graphical model for automatic code generation.

Background of the Invention

Various classes of block diagrams describe computations that can be performed on application specific computational hardware, such as a computer, microcontroller, FPGA, and custom hardware. Classes of such block diagrams include time-based block diagrams such as those found within Simulink® from the MathWorks, Inc. Natick Ma, state-based and flow diagrams such as those found within Stateflow® from the MathWorks, Inc. Natick Ma., data-flow diagrams and software diagrams, such as those found in the Unified Modeling Language. A common characteristic among these various forms of block diagrams is that they define semantics on how to execute the diagram.

Historically, engineers and scientists have utilized time-based block diagram models in numerous scientific areas such as Feedback Control Theory and Signal Processing to study, design, debug, and refine dynamic systems. Dynamic systems, which are characterized by the fact that their behaviors change over time, are representative of many real-world systems. Time-based block diagram modeling has become particularly attractive over the last few years with the advent of software packages such as Simulink from The MathWorks, Inc. of Natick, Massachusetts. Such packages provide sophisticated software platforms with a rich suite of support tools that makes the analysis and design of dynamic systems efficient, methodical, and cost-effective.

A dynamic system (either natural or man-made) is a system whose response at any given time is a function of its input stimuli, its current state, and the current time. Such systems range from simple to highly complex systems. Physical dynamic systems include a falling body, the rotation of the earth, bio-mechanical systems (muscles, joints, etc.), bio-chemical systems (gene expression, protein pathways), weather and climate pattern

systems, etc. Examples of man-made or engineered dynamic systems include: a bouncing ball, a spring with a mass tied on an end, automobiles, airplanes, control systems in major appliances, communication networks, audio signal processing, nuclear reactors, a stock market, etc. Professionals from diverse areas such as engineering, science, education, and economics build mathematical models of dynamic systems in order to better understand system behavior as it changes with the progression of time. The mathematical models aid in building “better” systems, where “better” may be defined in terms of a variety of performance measures such as quality, time-to-market, cost, speed, size, power consumption, robustness, etc. The mathematical models also aid in analyzing, debugging and repairing existing systems (be it the human body or the anti-lock braking system in a car). The models may also serve an educational purpose of educating others on the basic principles governing physical systems. The models and results are often used as a scientific communication medium between humans. The term “model-based design” is used to refer to the use of block diagram models in the development, analysis, and validation of dynamic systems.

Dynamic systems are typically modeled as sets of differential, difference, and/or algebraic equations. At any given instant of time, these equations may be viewed as relationships between the system’s output response (“outputs”), the system’s input stimuli (“inputs”) at that time, the current state of the system, the system parameters, and time. The state of the system may be thought of as a numerical representation of the dynamically changing configuration of the system. For instance, in a physical system modeling a simple pendulum, the state may be viewed as the current position and velocity of the pendulum. Similarly, a signal-processing system that filters a signal would maintain a set of previous inputs as the state. The system parameters are the numerical representation of the static (unchanging) configuration of the system and may be viewed as constant coefficients in the system’s equations. For the pendulum example, a parameter is the length of pendulum and for the filter example; a parameter is the values of the filter taps.

There are four common types of mathematical models used in the study of dynamic systems. The first type of mathematical model describes systems using ordinary differential equations (ODEs) and is depicted in Figure 1A. The dynamic system 2

specifies a set of two equations: Output 4 and Derivative 6. The Output equation 4 facilitates the computation of the system's output response at a given time instant as a function of its inputs, states, parameters, and time. The Derivative equation 6 is an ordinary differential equation that allows the computation of the derivative of the states at the current time as a function of the inputs, the states, parameters, and time. This class of models is suitable for systems in which it is important to track the system response as a continuous function of time. Such continuous-time systems are commonly representative of physical systems (mechanical, thermal, electrical). For simple systems, it may be possible to use the Output 4 and Derivative equations 6 to obtain a closed-form solution for the output response $y(t)$. But in most complex real world systems, the response of the system is obtained by integrating the states through numerical means.

The definition of an ODE used herein encompasses both implicit and explicit differential equations. The class of ordinary differential equations may require additional equations to define the system being modeled. For example, equations called projections may be required to impose constraints on the differential variables (e.g., states X_1 and X_2 must fall on the manifold defined by $X_1^2 + X_2^2 = 25$). These constraints can be either applied as a secondary condition or a coupled condition to the differential equation. Although systems including the projections may conventionally no longer qualify as an ODE; they are included here to simplify the categories of systems. Another example is the use of a Jacobian equation that defines partial derivatives with respect to the independent and/or differential variables. The Jacobian equation is typically used when obtaining a linear approximation of a non-linear model or an overall linear model of a set of equations. Jacobian equations are required for some forms of numerical integration, for producing the linear model once the model has reached its steady state operating point, etc. The Output 4 and Derivatives equations 6 may be extended to define other relationships for the block. For example, the Output equation 4 may help manage its states by defining a relationship where it resets the state back to a known quantity at a specific point in time or when a specific condition is seen.

Another type of mathematical model describes systems using difference equations as depicted in Figure 1B. The dynamic system 8 specifies a set of two equations: Output 10 and Update 12. The Output equation 10 facilitates the computation of the system's

output response at a given time instant as a function of the inputs, states at some previous time, parameters, and time. The Update equation 12 is a difference equation that allows the computation of the states at the current time as a function of the inputs, states at some previous time, parameters, and time. This class of models is suitable for systems in which it is important to track the system response at discrete points in time. Such discrete-time systems are commonly representative of discrete-time control and digital signal processing systems. For simple systems, it may be possible to use the Output 10 and Update equations 12 to obtain a closed-form solution for the output response $y(t)$. But in most complex real world systems, the response of the system is solved through recursion. The Output 10 and Update equations 12 are applied repeatedly to solve for the system response over a period of time.

An additional type of mathematical model describes systems using algebraic equations as depicted in Figure 1C. The dynamic system 14 uses an algebraic equation 16 that needs to be solved at each time to obtain the outputs. While simple systems may allow one to obtain a closed-form solution for the system inputs and outputs, practical algebraic equations may best be solved iteratively using a numerical method involving both perturbations and iterations. Algebraic equation solving techniques used in the context of dynamic system modeling are discussed in greater detail below.

A fourth type of mathematical model is a composite system that has components that fall into the three types of models discussed above. Most complex real-world system models fall into this category. This class of systems has Output, Derivative, Update, , and potentially other equations. Solving for the output response of such systems requires a combination of the solution approaches discussed for all of the classes above. One example of a composite system is one described by differential-algebraic equations (DAEs) which contain both differential equations and algebraic equations. Grouped within the composite class of systems are many extensions involving relationships (equations) defined in terms of both outputs and state. For example, one can define a limited integration relationship for a differential variable. This relationship requires a set of equations that consists of the Output equation, an Update equation, a Derivative equation, and a Zero Crossing equation. The Zero Crossing equation defines the points in time where the upper and lower limits of the limited integration occur.

Another example of an extension is the notion of Enable and Disable equations that define relationships among states or signals when parts of a system are activated and deactivated during execution.

Inherent in the four classes of systems (ODE, difference equations, algebraic equations and composite) is the notion of system sample time. The sample-time is the time interval at which the inputs, state, or outputs (collectively referred to as the results) of the system are traced as time progresses. Based on sample times, a system can be described as a discrete-time system, continuous-time system and hybrid system.

A discrete-time system is a system in which the evolution of the system results are tracked at finite intervals of time. In the limit as the interval approaches zero, the discrete-time system becomes a continuous-time system. The intervals of time may be periodic or non-periodic. Sometimes, non-periodic rate systems are referred to as non-uniform rate systems meaning that there is no periodic rate at which the response can be tracked

Nonuniform-rate systems can fall into the class of composite systems where an additional equation (GetTimeOfNextVarHit) defines when in the future the other equations associated with the system should be evaluated. A continuous-time system is a system in which the evolutions of the system results are continuously changing.

Continuous-time signals change during numerical integration (minor time steps). An example of a continuous-time system is one described by an ODE. There can also be algebraic or composite continuous-time systems. A hybrid system is a system with both discrete-time and continuous-time elements.

If a system has only one sample time, it is said to be single-rate. If a system has multiple sample times, it is said to be multi-rate. Multi-rate systems can be evaluated (executed) using either a single-tasking form of execution or a multi-tasking form of execution. When multi-tasking execution is used, it conforms to rate monotonic scheduling principals as defined by Liu, C. L., and LAYLAND, J. W. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. ACM 20, 1

(January 1973), 46-61. Systems may also be categorized by the type of numerical integration solver being used. A fixed-step system is one that uses a fixed-step solver. Fixed-step solvers typically use explicit methods to compute the next continuous state at fixed periodic intervals of time. A variable-step system is one that is using a variable-step

solver. A variable-step solver can use either implicit or explicit methods to compute the next continuous state at non-periodic intervals of time. Generally, variable-step solvers use a form of error control to adjust the interval size such that the desired error tolerances are achieved.

5

In practice, except for the most basic systems, mathematical models for dynamic systems involve a complex set of mathematical transformations applied in some prescribed manner with the outputs of some transformations forming the inputs of others. Each elemental transformation may be viewed in isolation as a simple dynamic system
10 falling into one of the categories listed above. Therefore, a complex dynamic system may be modeled as an interconnection of various simple dynamic systems. A schematic representation of such an interconnection that has evolved over the years is the block diagram. Such block diagram models have now become a standard means in textbooks, design papers, journal articles, and specifications to communicate the details of a dynamic
15 system's behavior.

A block diagram model of a dynamic system is represented schematically as a collection of blocks interconnected by lines that represent signals. A signal represents the input and output of a dynamic system. Each block represents an elemental dynamic
20 system. A line emanating at one block and terminating at another signifies that the output of the first block is an input to the second block. Each distinct input or output on a block is referred to as a port. Signals correspond to the time-varying quantities represented by each line connection and are assumed to have values at each time instant. The source block of a signal writes to the signal at a given time instant when its system equations are
25 solved. The destination blocks of this signal read from the signal when their system equations are being solved. The basic components of a block diagram are illustrated in Figure 2. The block diagram includes a plurality of blocks 20, lines 22 and ports 24 that are interconnected. Those skilled in the art will recognize that the term "blocks" does not refer exclusively to elemental dynamic systems but may also include other modeling
30 elements that aid in readability and modularity of block diagrams.

The theory of Digital Signal Processing (DSP) focuses on modeling signals as sequences of samples. This view naturally fits into the time-based block diagram

paradigm by mapping the samples $u[n]$ to discrete-time points $u(t_k)$. This adds the benefit of being able to model the interaction between DSP systems and other classes of time-based systems, e.g. continuous and/or discrete-time control systems.

Put another way, block diagram models are time-based relationships between signals and state variables representative of a dynamic system. The solution (computation of system response) of the model is obtained by evaluating these relationships over time, where time starts at a user-specified “start time” and ends at a user-specified “stop time”. Each evaluation of these relationships is referred to as a time step. Signals represent quantities that change over time, and these quantities are defined for all points in time between the block diagram’s start and stop time. The relationships between signals and state variables are defined by sets of equations represented by blocks. These equations define a relationship between the input signals, output signals, state, and time. Inherent in the definition is the notion of parameters, which are the coefficients of the equations.

It is important to note that block diagrams are not exclusively used for representing time-based dynamic systems but also for other models of computation. For instance, flow-charts are block diagrams used to capture process flow and are not generally suitable for describing dynamic system behavior. Data flow block diagrams are block diagrams that describe a graphical programming paradigm where the availability of data (often thought of as tokens) is used to initiate the execution of blocks, where a block represents an operation and a line represents execution dependency describing the direction of data flowing between blocks. As used herein, the term block diagrams means time-based block diagrams used in the context of dynamic systems except as otherwise noted.

Block diagram modeling has spawned a variety of software products such as Simulink from The Mathworks, Inc. of Natick, Massachusetts, that cater to various aspects of dynamic system analysis and design. Such products allow users to perform various types of tasks including constructing system models through a user-interface that allows drafting block diagram models, allowing augmentation of a pre-defined set of blocks with custom user-specified blocks, the use of the block diagram model to compute and trace the temporal evolution of the dynamic system’s outputs (“executing” the block

diagram), and automatically producing either deployable software systems or descriptions of hardware systems that mimic the behavior of either the entire model or portions of it (referred to herein as “code generation”). Each of the tasks listed above has many intricate details and subtle variations which are explored further below.

5

Block modeling software includes a number of generic components. Although the discussion contained herein focuses on Simulink version 5.0 (Release 13) from the MathWorks, Inc. of, Natick MA, those skilled in the art will recognize that it is applicable to other block modeling software applications. The generic components include a block diagram editor, blocks and a block diagram execution engine. The block diagram editor allows users to perform such actions as draw, edit, annotate, save, and print out block diagram representations of dynamic systems. As noted earlier, blocks are the fundamental mathematical elements of a classic block diagram model. Simulink extends the classic block diagram models by introducing the notion of two classes of blocks, non-virtual blocks and virtual blocks. Non-virtual blocks are elementary dynamic systems. A virtual block is provided for graphical organizational convenience and plays no role in the definition of the system of equations described by the block diagram model. Examples of virtual blocks are the Bus Creator virtual block and Bus Selector virtual block which are used to reduce block diagram clutter by managing groups of signals as a "bundle". Virtual blocks may be used to improve the readability of models. Simulink further extends the meaning of a non-virtual block to include other semantics, such as a “merge” block semantic. The merge block semantic is such that on a given time step its output is equal to the last block to write to an input of the merge block. An additional extension provided by Simulink is the concept of conditional execution. Simulink contains the concept of conditional and iterative subsystems that control when in time block methods execute for a sub-section of the overall block diagram.

A block diagram execution engine contributes to the modeling software task of enabling the computation and tracing of a dynamic system’s outputs from its block diagram model. An execution engine carries out the task of compiling and linking the block diagram to produce an “in-memory executable” version of the model that is used for generating code and/or simulating or linearizing a block diagram model. Note that execution of the block-diagram is also referred to as simulation. The compile stage

involves checking the integrity and validity of the block interconnections in the block diagram. In this stage, the engine also sorts the blocks in the block diagram into hierarchical lists that are used when creating the block method execution lists. In the link stage, the execution engine uses the result of the compiled stage to allocate memory
5 needed for the execution of the various components of the block diagram. The linking stage also produces block method execution lists, which are used by the simulation or linearization of the block diagram. Included within the link stage is the initialization of the model which consists of evaluating “setup” methods (e.g. block start, initialize, enable, and constant output methods). The block method execution lists are generated
10 because the simulation and/or linearization of a model must execute block methods by type (not by block) when they have a sample hit.

After linking has been performed, the execution engine may generate code. In this stage, the execution engine may choose to translate the block diagram model (or portions
15 of it) into either software modules or hardware descriptions (broadly termed code). If this stage is performed, then the stages that follow use the generated code during the execution of the block diagram. If this stage is skipped completely, then the execution engine uses an interpretive mode of execution for the block diagram. In some cases, the user may not proceed further with the execution of the block diagram because they would
20 like to deploy the code outside the confines of the block diagram software. Upon reaching the simulation stage, the execution engine uses a simulation loop to execute block methods in a pre-defined ordering upon a sample hit to produce the system responses they change with time.

25 For linearization, Simulink uses the block method execution lists in a prescribed fashion to produce a linear state space representation of the dynamic system described by the block diagram.

The block diagram editor is the graphical user interface (GUI) component that
30 allows drafting of block diagram models by a user. In Simulink, there is also a textual interface with a set of commands that allow interaction with the graphical editor. Using this textual interface, users may write special scripts that perform automatic editing operations on the block diagram. A user generally interacts with a set of *windows* that act

as canvases for the model. There is generally more than one window for a model because models may be partitioned into multiple hierarchical levels through the use of subsystems (discussed further below).

5 A suite of GUI tools in Simulink allows users to draft a block diagram model on the corresponding windows. The GUI tools include a block palette, wiring line connection tool, annotation tool, formatting tool, attribute editing tool, save/load tool and publishing tool. The block palette is a *library* of all the pre-defined blocks available to the user when they are building the block diagram. Individual users may be able to customize this
10 palette to: (a) reorganize blocks in some custom format, (b) delete blocks they do not use, and (c) add custom blocks they have designed. The palette allows blocks to be dragged through some human-machine interface (such as a mouse or keyboard) from the palette on to the window (i.e., model canvas). The graphical version of the block that is rendered on the canvas is called the icon for the block. There may be different embodiments for the
15 block palette including a tree-based browser view of all of the blocks.

The wiring line connection tool allows users to draw directed lines that connect the ports of blocks in the model's window. Lines are also added through various mechanisms involving human-machine interfaces such as the mouse or keyboard.

20 Simulink also provides various forms of auto-connection tools that connect blocks automatically on user request to produce an aesthetically pleasing layout of the block diagram (especially those with high complexity with large numbers of blocks). The annotation tool allows users to add notes and annotations to various parts of the palette for a block diagram. The formatting tool enables users to perform various formatting
25 operations that are generally available on any document editing tool. These operations help pick and modify the various graphical attributes of the block diagram (and constituent blocks) such as include font-selection, alignment & justification, color selection, etc. The block diagram and all the blocks within the block diagram generally have a set of functional attributes that are relevant for the execution or code-generation.

30 The attribute editing tool provides GUIs that allows these attributes to be specified and edited. The save/load tool allows a created block diagram model to be saved. The saved model can be reopened in the editor at some later juncture through a load mechanism. Simulink also allow users to save blocks including pre-constructed subsystems into a

separate class of block-diagrams called libraries. Such libraries facilitate reuse of the same block in a number of other block-diagrams. The load/save mechanism is specially equipped to handle loading and saving of blocks in a block-diagram that actually reside in libraries.

5

The publishing tool enables the viewing of the block diagram as a document that can be published in any of the standard document formats (examples: PostScript, PDF, HTML, etc.). Those skilled in the art will recognize that the windows for multiple models and all of the tools mentioned above could potentially be embedded in a single

10 Multi-Document Interface (MDI) for providing a unified software environment.

Those skilled in the art will also recognize that block-diagram packages offer scripting languages for writing out programs that automatically carry out a series of operations that would normally require interaction with the GUI. For example, Simulink offers a set of commands in MATLAB for carrying out operations such as block addition (*add_block*),
 15 block deletion (*delete_block*), starting and terminating execution (*set_param*), modifying block attributes (*set_param/get_param*), etc.

Simulink also offers a variety of other GUI tools that improve the ability of users to build and manage large block diagrams. Examples of such GUIs include: (a) a Finder
 20 that helps find various objects such as blocks and lines within a block-diagram, (b) a Debugger that helps debug the execution of block-diagrams, (c) a Revision Control UI for managing multiple revisions of the block-diagram, and (d) a Profiler for viewing timing results while executing a block-diagram.

25 A typical base data-structure for a block may be represented as:

```

class Block {
  public:
    // Access methods for setting/getting block data
  30    ...
    // Methods for block editing
    virtual ErrorStatus BlockDrawIcon();
    virtual BlockParameterData BlockGetParameterData();
    ...
  35    // Methods for block compilation
    ...
  
```

```

// Methods for block execution
.....
virtual ErrorStatus BlockOutput()    = 0;
virtual ErrorStatus BlockDerivative() = 0;
5   virtual ErrorStatus BlockUpdate()  = 0;
    ...
private:
    BlockGraphicalData blkGraphicalAttributes;
    BlockFunctionalData blkFunctionalAttributes;
10   BlockCompiledData  blkCompiledAttributes;
    BlockExecutionData blkExecutionData;
    ...
};

```

Although the example of the data structure above is written in C++, those skilled in the art will recognize that equivalent data structures written in other languages may also be used. The major data fields of the block data structure fall into four categories, a graphical attributes field, a functional attributes field, a compiled attributes field and an execution data field.

The graphical attributes field is responsible for storing information relevant for graphical rendering of the block within its parent block diagram's GUI. Attributes specific to the block icon such as font, color, name, and icon-image are stored in this field. It should be noted that modifying these attributes does not affect the dynamics of the model using this block. The functional attributes field is responsible for specifying block attributes that may potentially affect the dynamics of the model using this block. These attributes are specified for the block as a whole and the input and output ports of the block. Examples of block attributes include block sample times and restrictive flags. Block sample times specify if the block corresponds to an elemental, continuous, discrete, or hybrid dynamic system. If the block is an elemental discrete-time system, then the attribute specifies the spacing between time instants at which the block response should be traced. A restrictive flag disallows the use of blocks in certain modeling contexts. For example, one may impose the restriction that there may only be one instance of given block in a model.

Attributes of block ports specify properties of the data that is either available or produced at that port. Block port attributes include dimensions, datatypes, sample rates, and direct feedthrough. Dimension attributes are individual dimensions of a multi-

dimensional matrix that is used as a container for data elements. Datatype attributes are the datatype of each element of data in the data container. A complexity attribute is a flag to specify if each data element is real or complex. A sample rate attribute specifies how when the signal corresponding to an input or output port will be used. The port sample
5 times may sometimes be used to implicitly infer the block's sample time. The direct feedthrough attribute is specified only for input ports and indicates whether or not the Output and/or GetTimeOfNextHit equations of the block are a function of the given input. This attribute helps in determining the sequence in which block methods should be executed while executing the block diagram.

10

The compiled attributes field of the block data structure holds the attributes of the block and its ports that mirror the functional attributes listed above. This field is filled in during block diagram compilation by utilizing the functional attributes of the block in conjunction with the functional and compiled attributes of the blocks that are connected
15 to it. This process of determining the compiled attributes from the functional attributes is termed attribute propagation. Attribute propagation is described in greater detail below in the section on block diagram compilation. The execution data field is mainly responsible for storing the memory locations that are going to serve as sources for block inputs, outputs, states, parameters, and other work areas during execution of blocks.

20

The block data structure also has a set of associated methods that may be categorized as access methods to data fields, methods used in editing, methods used in compilation and methods used in execution. Access methods to data fields help in setting and getting the various data fields of the block. Methods used in editing are called
25 by the block diagram editor in order to render the block appropriately in the GUI of its parent block diagram. For instance, this set of methods may include a BlockDrawIcon method that determines the shape the block icon has on the GUI. Methods used in compilation are methods that are called by the block diagram compilation engine. They help validate the connections of the block to other blocks on the block diagram. The
30 methods used in execution include a number of different run-time methods that are required for execution. These include the BlockOutput, BlockUpdate, BlockDerivative methods that realize the Output, Update, and Derivative equations discussed earlier in the context of dynamic systems. In addition, to these methods Simulink includes several other

run-time methods, such as the Jacobian, Projection, ZeroCrossings, Enable, Disable, Initialize, EvalParams (check and process parameters), and GetTimeOfNextHit methods. It should be noted that there is no explicit method for algebraic equations because these are represented and processed in a different manner which will be discussed below.

5

The base data structure for the block specifies the generic fields and interfaces that need to be supported by a block. Some of the methods are purely virtual and have no specific implementation in the base block class. In order to define a specific block (such as an Integrator block), one needs to subclass the base block class and provide explicit definitions for these virtual methods. An example of the subclassing of a block may be seen by examining an Integrator block. Figure 3 depicts the desired behavior of an Integrator block 30. In order to create the subclass, four major categories of information within the subclass must be specified, the block parameters, the methods used in editing, the methods used in compilation, and the methods used in execution. The elemental dynamic system embodied by the block may be parameterized as illustrated in Figures 1A-1C. Each block needs to be able to specify its list of expected parameters. The block diagram editor's Attribute-Editing tool may allow users to specify the parameters for the block when they use it in their models. In the Integrator block example, the block has one parameter that specifies the block's initial condition for the block's state. Regarding the methods used in editing, the subclass needs to specify a method that renders its icon. For example, the Integrator block may implement a method that makes its icon be a box with a '1/s' within the box. Also, the subclass needs to instantiate a method that allows access of the block parameters from the GUI's Attribute-Editing tool. For the Integrator example, this method would allow users to specify the Initial Condition parameter on a GUI for the block. For the methods used in compilation, the subclass needs to instantiate methods that help in the compilation of the block diagram model in which it is placed. These methods help specify the compiled information for the inputs and outputs of the block. For instance, the Integrator block may specify a method that ensures that if the input to the Integrator is a vector, then the output is a vector of the same size. For methods used in execution, the subclass needs to instantiate specific Output, Derivative, and Update methods that represent the block behavior. In the case of the Integrator block, an Output and Derivative method are needed. It should be noted that in Simulink the Integrator block has additional methods that are not illustrated here. The Output method

sets the output to be equal to the state. The Derivative method sets the derivative of the state to be equal to the input.

The specification of these four types of information for the Integrator block subclass may be shown by a reduced form of the Simulink Integrator block:

```

IntegratorBlock : public Block {
public:
    10      ErrorStatus BlockDrawIcon() {
           // Draw '1/s' on the icon
           .....
        }
        BlockParameterData BlockGetParameterData() {
           // Return initial_condition as block data
        15      .....
        }
        ErrorStatus BlockOutput(){
           // Implement  $y(t) = x(t)$ 
           .....
        20      }
        ErrorStatus BlockDerivative(){
           // Implement  $dx(t)/dt = u(t)$ 
           .....
        }
        25      private:
           double initial_condition;

};

```

30 It should be noted that block diagram software generally provides open access to the block's data structure to users of the software. This allows users to create and utilize custom block implementations in their models.

Blocks in a block diagram may be virtual or non-virtual. The designation of a
 35 block as non-virtual indicates that it influence the equations in the mathematical model for the dynamic system. In the context of block diagram software, it is beneficial to include other *virtual* blocks that *do not* affect the equations in the dynamic system's model. Such blocks help improve the readability and modularity of the block diagram and wield no semantic influence on the mathematical model. Examples of such virtual blocks
 40 include virtual subsystems, inport blocks and outport blocks, bus creator blocks and From and Goto blocks.

Modularity may be achieved in a block diagram by layering the block diagram through the use of subsystems. A subsystem facilitates layering by allowing a collection of blocks to be represented by a single block with input and output signals. The input and output signals of the subsystem are accessible to the constituent blocks within the subsystem. A subsystem is a virtual subsystem if its constituent blocks are moved back into the main block diagram model during the model's execution. Within a virtual subsystem graphical entities, called inport and output blocks, are provided to define signal connections to the parent block diagram. These inport and output blocks indicate a tunnel-through signal connection to the parent block diagram.

Additional types of virtual blocks include bus creator blocks and selector blocks. In large models, there may be an extensive set of lines that connect one section of a block diagram to another section. To avoid excessive clutter of lines and improve readability, there is typically a special block called a *Bus Creator* that helps bundle all of the lines together to form a single *bus line*. This single bus line then connects the two sections of the model. At the destination end of the line, a block called a *Bus Selector* helps unbundle the individual lines so that they can be connected to other blocks.

Other virtual blocks include From blocks and Goto blocks that are special blocks that help avoid graphical clutter, e.g. a line that connects two distant sections of a block diagram. The line is terminated close to its originating point by a From block. At the other end, a new line is drawn from a Goto block that is hot-linked to the From block. Each Goto and From block has an associated tag that describes which blocks are connected together. An important point to be noted is that Virtual blocks have neither execution data nor execution methods in their data structure.

Simulink also provides the user with the ability to extend the simulator by providing the ability to enhance the simulator with blocks that define dynamic systems or are virtual properties. The extension is provided through a language independent API (e.g. C, C++, Ada, Fortran, Assembly, M).

As noted previously, to facilitate modeling fairly large and complex dynamic systems, Simulink allows users to layer their block diagrams. A subsystem facilitates such layering by allowing a collection of blocks to be represented by a single block with input and output signals. The input and output signals of the subsystem are accessible to its constituent blocks. By nesting subsystems within each other, one can create block diagrams with arbitrary layers of hierarchy. Ideally a subsystem has no impact on the meaning of the block diagram. Additionally, subsystems provide a way of grouping blocks together and allowing other block diagram constructs to impose unified control on the constituent blocks. To enhance the modularity of subsystems, modeling software also allows aggregated list(s) of parameters of the blocks within the subsystem to be accessed from a single GUI, and defines and displays special icons on the subsystems. The process of defining the parameter list and the special icon is called masking a subsystem.

There are two main types of subsystem blocks, virtual subsystems and non-virtual subsystems. Virtual subsystems serve the purpose of providing the block diagram with a graphical hierarchy. Non-virtual subsystems behave like an elemental dynamic system with its own execution methods (Output, Update, Derivatives, etc.). These execution methods in turn call the execution methods of the constituent blocks.

The classes of non-virtual subsystems are:

Atomic subsystems. These are similar to virtual subsystems, with the advantage of grouping functional aspects of models at a given layer. This is useful in modular design.

Conditionally-executed subsystems. These are non-virtual subsystems that execute only when a precondition is fulfilled:

Enabled subsystems. These are similar to Atomic subsystems, except that the constituent blocks only execute when an enable signal feeding the subsystem is greater than zero.

Triggered subsystems. These are similar to Atomic subsystems, except that the constituent blocks only execute when a rising and/or falling signal is seen on a triggering signal feeding the subsystem.

Enable with Trigger subsystems. These are an intersection of the properties of Enabled and Triggered subsystems.

Action subsystems. These subsystems are connected to action-initiator (e.g., an “If” or “SwitchCase” block), a block that explicitly commands the subsystem contents to execute. These subsystems are similar to Enabled subsystems except that the management of the “enabling” signal has been delegated to an action-initiator. Action subsystems

5 define a new type of signal, called an action signal that signifies which subsystems are commanded to execute by the action-initiator.

Function-call subsystems. These subsystems provide a means of collecting blocks into a subsystem that is only executed when called by an owner block. The owner block may compute input signals for the subsystem before calling the subsystem. Additionally, the

10 owner may also read output signals from the subsystem after calling it. Function-call subsystems define a new type of execution control signal, called a function-call signal that contains no data. It is used to define the execution relationship between the owner block and the function-call subsystem. Function-call owners may also designate themselves as an “interrupt” source. In simulation, they simulate the effects of an

15 interrupt and in code generation they can attach themselves to an (asynchronous) interrupt.

While subsystems and For subsystems. These subsystems execute the constituent blocks multiple times on a given time step.

20 Simulink allows for several forms of block parameters to be defined. There are two general categories of parameters: those parameters that can be modified during simulation and those that cannot be modified. An example of a parameter that may be modified during simulation is the amplitude of a Sine Wave block if configured by the user to allow modification during execution. A parameter such as the amplitude specifies

25 coefficients of the dynamic equation, in this case the amplitude of the sine wave function defined by the Sine Wave block. An example of a parameter that can never be modified during simulation is the sample time of the Sine Wave block. The parameters that can be modified during simulation are further broken down into other categories which include mapping the dialog parameter (e.g. the amplitude) to run-time parameters or converting

30 the dialog parameter to an inlined (non-modifiable) parameter. Run-time parameters can further be mapped to mathematical expressions of tunable Matlab variables or Matlab parameter objects describing properties of the variables (called Simulink.Parameter's). A

global run-time parameter data structure is used within Simulink to manage the block parameters during the execution of the model.

In addition to block parameters, there are model-wide parameters which are generally associated with the solver. These parameters include aspects such as the time span in which to perform a simulation, the type of solver, and the time span. Simulink gives the user the ability to adjust solver parameters during model execution. The adjustment of these solver parameters is performed at the start of a time step.

Once a block diagram model has been constructed using the editor, an execution engine allows the model to be solved in order to trace the system outputs as a function of time. The solution of the model, which may be referred to as model execution, is carried out over a user-specified time span for a set of user-specified inputs. Simulation proceeds in four major stages: compilation, link, code generation, and the simulation loop.

Alternatively, the execution engine can obtain a linear representation of the model (linearization). The interrelationship between the various stages is illustrated in a flowchart in Figure 4.

The execution begins when the block diagram is compiled. Following the compilation stage, is the model link stage which may also produce linear models. Code may or may not be generated. If code is generated, a decision is made whether to continue the simulation. If the decision is made to continue the simulation the model is simulated/executed through the Simulation Loop. If the simulation is not continued, the code may be delivered to a target and executed in an external mode. If code is not generated the block diagram may execute in interpretive mode when entering the Simulation Loop.

The compile stage marks the start of model execution and involves preparing data structures and evaluating parameters, configuring and propagating block characteristics, determining block connectivity, and performing block reduction and block insertion. The preparation of data structures and the evaluation of parameters creates and initializes basic data-structures needed in the compile stage. For each of the blocks, a method forces

the block to evaluate all of its parameters. This method is called for all blocks in the block diagram. If there are any unresolved parameters, execution errors are thrown at this point.

During the configuration and propagation of block and port/signal characteristics, the compiled attributes (such as dimensions, datatypes, complexity, or sample time) of each block (and/or ports) are setup on the basis of the corresponding functional attributes and the attributes of blocks (and/or ports) that are connected to the given block through lines. The attribute setup is performed through a process during which block functional attributes “ripple through” the block diagram from one block to the next following signal connectivity. This process (referred to herein as “propagation”), serves two purposes. In the case of a block that has explicitly specified its block (or its ports’) functional attributes, propagation helps ensure that the attributes of this block are compatible with the attributes of the blocks connected to it. If not, an error is issued. For instance, if an Integrator block is implemented to only accept numbers of double precision datatype, then this block will error out if it is driven by a block that produces single precision data, unless the user has asked for an implicit data conversion. Secondly, in many cases blocks are implemented to be compatible with a wide range of attributes. Such blocks adapt their behavior in accordance with the attributes of the blocks connected to them. This is akin to the concept of polymorphism in object-oriented programming languages. For instance, a discrete-time Filter block could be implemented to accept any of the standard integer datatypes ranging from 8-bit to 128-bit. The exact implementation of the block is chosen on the basis of the specific block diagram in which this block finds itself. Included within this step are other aspects such as validating that all rate-transitions within the model yield deterministic results and that the appropriate rate transition blocks are being used.

25

The compilation step also determines actual block connectivity. Virtual blocks play no semantic role in the execution of a block diagram. In this step, the virtual blocks in the block diagram are optimized away (removed) and the remaining non-virtual blocks are reconnected to each other appropriately. This compiled version of the block diagram with actual block connections is used from this point forward in the execution process

30

Once actual block connectivity has been determined (by removing the virtual blocks) the block diagram may be further optimized by performing block reduction and

insertion. During this step, non-virtual blocks may be inserted or a set of non-virtual blocks may be completely removed or reduced to a single equivalent block. Block insertion and reduction is mainly done to improve execution efficiency. Examples of block insertion and reduction include the removal of Gain blocks whose gain value is 1.

- 5 A Gain block is a block which multiplies its input value by a gain parameter, such as a simple amplifier. Figure 5 depicts the replacement of a collection of blocks 60, 62, and 64 connected in an accumulator pattern and leading to result 66 with an equivalent synthesized block 68 representing the accumulator pattern leading to the same result 66. A signal copy block may also be automatically inserted in order to make contiguous
- 10 memory copies of signals that are made up of disjoint memory sections. Block insertion and reduction may also be performed at other suitable stages of compilation.

- The way in which blocks are interconnected in the block diagram does not necessarily define the order in which the equations (methods) corresponding to the
- 15 individual blocks will be solved (executed). The actual order is partially determined during the sorting step in compilation. Once the compilation step has completed, the sorted order cannot be changed for the entire duration of the block diagram's execution.

- The first step in sorting involves transforming the graphical block diagram into a
- 20 compiled (in-memory) directed graph consisting of arcs and vertices. The vertices are derived from some of the non-virtual blocks. For instance, virtual and reduced blocks do not appear in the directed graph. The arcs represent data dependencies between the vertices. The data dependencies do not correspond to the signals in the block diagram. For example, all signals that connect to input ports without direct feed through are "cut" or
- 25 ignored. In addition, data dependencies are added to capture implicit dependencies. For example, all inputs to a Function-Call subsystem are implicit data dependencies to the owner (caller) block.

- The process of converting a block diagram into a compiled directed graph is
- 30 shown in Figure 6A. A block diagram 81 includes a Sine Wave 1 block 82, a Sine Wave 2 block 84, a Goto block 86, a Function Call Generator block 88, and a From block 90. Also included are a Function Call Subsystem block 92, a Sum block 94, a Gain block 96, an Integrator block 98 and an Output (Output 1) block 100. Those blocks which are not

virtual or reduced appear on the corresponding directed graph 111. The directed graph 111 includes a Sine Wave1 vertice 112, a Sine Wave 2 vertice 114, a function-call generator vertice 116, and a function call subsystem vertice 118. Also included are a Sum vertice 121, a Gain vertice 122, an Integrator vertice 124 and an Outport 1 vertice 126. The vertices are connected by arcs.

The graph is used to sort the blocks into a linear sorted list . Figure 6B depicts a sorted list 128 generated from the compiled directed graph 111 which includes the elements appearing as vertices in the directed graph 111 sorted into order. The root block diagram has a sorted-list associated with it. Roughly speaking, each non-virtual subsystem layer and some special block diagram elements also each have their own sorted-list. During the sorting of the graph into the list, strongly connected components are identified. The term strongly connected section, which is a term that originates from graph theory, is a subset, S, of the blocks of a block diagram such that any block in S is reachable from any other block in S by following signal connections and S is not a subset of any larger such set. Strongly connected sections are flagged as algebraic loops when all blocks have direct feedthrough (an example is shown in Figure 6A consisting of the Sum 121 and Gain 122 blocks). Such loops correspond to a set of algebraic equations and are solved using iterations and perturbations during block diagram execution by solving for the algebraic variables. Algebraic variables are either specified by the user via Initial Condition blocks or chosen by the execution engine. Solving of algebraic loops is discussed further below.

Sorting must also take into consideration other user specified dependencies between the blocks. These dependencies include the concepts of priorities and placement groups. A block priority specifies the order in which the equations associated with a block are evaluated with respect to other blocks. Placement groups are a way of causing each class of block methods for a specified set of blocks to be “placed together” in the block method execution lists. The terms “data dependency” or “data precedence” as used herein refers to the arcs of the compiled directed graph and not the signals found within a block diagram. Attempting to correlate data dependencies directly to the signals found within a block diagram is incorrect and leads to the conclusion that Simulink does not satisfy data

dependencies, i.e., the execution of the operations or block methods does not satisfy data dependencies if one interprets signal connectivity as specifying data dependencies.

After compilation, the link stage commences. During this stage physical memory
5 allocations are made in order to prepare for execution. Buffers are allocated for block
input and output data buffers, states, and work areas. Additionally, block method
execution lists that are derived from the sorted list allow for execution of the block
diagram. Each block method execution list is a list of block methods that are to be
executed in a sequence when each method within the list has a sample hit. There is
10 generally a set of block method execution lists associated with each layer of the block
diagram that corresponds to a non-virtual subsystem. Non-virtual subsystems are either
defined by the user or automatically synthesized during compilation to either efficiently
execute the model or simplify the implementation of the semantics defined by Simulink.
In multi-tasking mode, the lists within each layer may be further partitioned when block
15 diagrams have blocks with different sample rates. These lists are explained in greater
detail below.

Those skilled in the art will recognize that while the block method execution lists
are derived from the sorted list, they do not necessarily correspond one-to-one with the
20 sorted lists. First, each block method execution lists contains only blocks that have such a
block method of the given type (class) defined by the list. Second, block methods
corresponding to components like the function-call subsystem do not appear on the block
method execution lists because they are executed by an “owner” block.

25 Although included in the discussion of the compilation stage, it is not required that
the time-based diagram perform the block sorting step during compilation. The sorting
step is performed to achieve efficient execution. Ignoring efficiency, there is no semantic
reason to perform the sorting step. Any random ordering of the block methods will work.
In fact, any ordering of all block method execution lists except the Output block method
30 execution list will result in the same level of efficiency. Randomly re-ordering the Output
block method execution list will yield correct answers. If the Output block method list is
randomly ordered, then the Simulation engine, when executing the Output block method

execution list, continues sequencing through the Output block method execution list at each point in time until there are no changes.

Similarly included within the linking stage for the sake of simplicity, is the memory initialization of the model. The memory initialization of the model includes invoking block start, initialize, constant initialize, enable, and constant output methods. These are examples of some of the block methods that are used during model setup (prior to execution) to initialize the “state” of the system so that execution or linearization can commence.

The compiled and linked version of the block diagram may be directly utilized to execute the model over the desired time-span. This interpretive mode of execution is suitable for getting fine-grained signal traceability. It should be noted that the traceability associated with interpretive execution comes at the price of increased overhead in the form of additional execution-related data-structures and messaging in the engine. An alternative to the interpretive execution mode is to utilize the generated-code created by Real-Time Workshop tool for Simulink models. In this mode, the engine (upon the behest of the user) translates a selected portion of the block diagram (or the entire block diagram itself) into code. Such code could be in a number of possible forms. The code may be instructions in a high-level software language such as C, C++, Ada, etc., hardware descriptions of the block diagram portions in a language such as HDL, or custom code formats suitable for interpretation in some third-party software. Alternatively, the code may be instructions suitable for a hardware platform such as a microprocessor, microcontroller, or digital signal processor, etc., a platform independent assembly that can be re-targeted to other environments, or just-in-time code (instructions) that corresponds to sections of the block diagram for accelerated performance.

The execution of a portion of the block diagram represented in code may be performed in a number of different ways based on the specific code format. The portion of the block diagram may execute a compiled version of the code generated in a high-level language (accelerated or software-in-the-loop simulation), the execution may simulate code that corresponds to a hardware description on a hardware simulator, (co-

simulation execution), the execution may involve calling out to third-party software to run code generated for such software (co-simulation execution), or the execution may call out directly to hardware that will run code that was generated and compiled for that hardware (processor-in-the-loop execution).

5

There are several different advantages to execution through code generation: Execution of generated code can be more efficient than interpretive execution because of fewer data-structures and lesser internal messaging in the engine, although the increased efficiency generally comes at the cost of decreased execution traceability. Simulation of hardware descriptions during execution can help identify and resolve bugs in the software stage of a design project. Such bugs prove much more expensive to track and fix once the system has been implemented in hardware. Additionally, block diagram modeling software can be integrated with other software environments that are suitable for modeling and simulating special classes of systems. Models can be tested directly in hardware thereby making prototyping of new systems fast and cost-effective. For instance, consider the design of a controller for an anti-lock braking system of a car. The dynamics of the braking system can be executed in the interpretive mode in the block diagram. The controller itself can be implemented on a hardware micro-controller to test the efficiency of the control laws implemented within. Note that for such target execution, it is normally necessary for the time span over which a model is executed by the software to match real-world time. In other words, the software must allow real-time execution of the block diagram model. Those skilled in the art will recognize that when users generate code, they may choose to not proceed further with the block diagram's execution. They may choose to take the code and deploy it outside of the confines of the modeling software environment. This is normally the last step in the design of dynamic systems in a block diagram software package.

There are several forms of target code execution known to those skilled in the art such as Rapid Prototyping, Embedded System Deployment, and Hardware-in-the-Loop which execute a model or portions of a model via the generated code on a Real-Time System target. One aspect of deploying (executing) the generated code on a target is the notion of "external mode." External mode refers to a system where Simulink acts as a monitor and debugger of the generated code running in real-time on a target. In External

Mode, users can change parameters and view signals via standard Simulink elements. Another important aspect of the code generation technology is that it is very extensible. Provided with the Simulink product family is the Target Language Compiler (TLC). This technology enables the creation of “active scripts” that control how the generated code is produced for a block diagram. Using TLC, one can tailor the generated code to suite their specific needs.

The execution of the block diagram uses a Simulation Loop (SimLoop) for solving for the block diagram’s outputs for a specified set of inputs over a specified span of time (“Time” in reference to the Simulation Loop means the time-line corresponding to the tracing of the dynamic system’s outputs, not real-world time unless otherwise noted). The term “SimLoop” applies to real-time systems where each iteration is tied to a physical periodic clock or other timer source. During this process, the block methods (equations) corresponding to the individual blocks are executed by type following their sorted order when they have a sample hit. The term “block execution” is loosely used to mean executing all block methods associated with the given block for a given time step, generally starting with the output method. Strictly speaking, blocks do not execute; the engine executes (evaluates) the appropriate block methods at the appropriate time points.

SimLoop has two variants “single-tasking” and “multi-tasking” depending on sample times. In general, the sample time of a block is the interval of time between calls to the Output, Update, and/or Derivative methods for a given block. In computing this interval, repeated calls at the same time instant (not in real-world time but the time corresponding to the execution of the dynamic system) are counted as the same call. A block’s sample rate may also be thought of the interval between successive executions of the block methods. If there is no uniform or regular interval between calls, then the block is said have a continuous sample time. If a uniform time interval can be found, then the block is said to have a discrete sample- time equal to that interval. Although blocks may be associated with more than one sample time in a sufficiently complex dynamic system the descriptions contained herein are confined to blocks with a single sample-time. Those skilled in the art will recognize that the descriptions may be extended to encompass blocks with multiple sample times.

Figure 7A depicts an abstract example of a block diagram being executed. The diagram includes a plurality of blocks 141, 142, 144, 146, 148 and 149. The block ports that have direct feedthrough are explicitly marked (using the symbol 'df')

Additionally, an abstract view of the execution methods instantiated by each block is shown in Figure 7B. The blocks contain a number of different methods 161, 162, 164, 166 and 168. Execution methods includes the three basic execution methods discussed earlier: Output, Update, Derivative, as well as several other methods that aid in advanced block functions such as initialization, linearization and zero-crossing detection.(which are discussed below). The data-dependencies between the compiled vertices created during sorting are used to generate the Sorted List 171 shown in Figure 7C.

A block diagram consisting of blocks that all have the same sample time is said to correspond to a single-rate system. A block diagram consisting of blocks that have more than one sample time corresponds to a multi-rate system. Figure 8 depicts a multi-rate system, adding sample-time information to the block diagram of Figure 7A. The plurality of blocks 141, 142, 144, 146, 148, and 149 each have an associated sample time. Since the sample times in the block diagram differ between blocks, the system is considered a multi-rate system. Block A 141, block E 148 and block F 149 each have a sample time of 0.1 seconds. Block B 142, block C 144 and block D 146 each have a sample time of 1.0 seconds.

The SimLoop is the heart of the execution engine. Each full pass through the loop is responsible for computing the outputs of the system at a particular time. At the end of each loop, the execution time corresponding to the next pass through the loop is computed. If this time exceeds the stop time specified by the user, the execution terminates. Within the loop, the sequence in which individual block equations are solved is determined by two pieces of information: the sample times of the blocks and the sorted order determined during the Compile stage. The amalgamation of these two pieces of information gives the execution lists for the block diagram. Those skilled in the art will recognize that the execution lists are created in the Link stage and are explained in the context of SimLoops for convenience. There are two distinct approaches for building execution lists and using them in the SimLoop. These approaches correspond to the

Single-tasking and Multi-tasking SimLoops summarized in the discussion on Figure 10 below.

Simulink also has the ability to modify coefficients (parameters) of blocks who declare their parameters as tunable. An example of a block is a Sine Wave block that implements the function $\text{output}(\text{time}) = \text{Amplitude} * \sin(\text{frequency} * \text{time} + \text{phase}) + \text{bias}$, where time is the independent variable and the parameters are: amplitude, frequency, phase, bias. When these parameters are declared as tunable, Simulink lets the user change these coefficients during simulation. Changing parameters is a drastic operation in that the definition of the model has changed (e.g. the sine block defines equations that describe the system). Thus, to enable the changing of parameters during the SimLoop, Simulink first queues parameter changes and then applies them on the next time step. Thus, the changing of parameters is not immediate. The delay in the changing of parameters is needed to ensure system stability. The application of the parameters at the start of the next time step is combined with the reset of the solver (Integrator) if needed.

For the purpose of exploring single-task loops and multi-task loops, Figure 9 depicts the block diagrams of Figure 7A and Figure 8 where Method1 corresponds to the Output method 189 and Method2 corresponds to the Update method 192. All other methods are ignored in the explanation of the loops. Simpler loops which do not include blocks that have continuous sample times are used in the example since the explanation is simpler in the context of discrete sample times and it is straight-forward to extend to continuous sample times.

In a single-tasking SimLoop, there is essentially a single execution time-line. On this time-line, each block is executed when it has a sample hit. A sample hit is defined to an execution time instant that is an integer multiple of the block's sample time. To aid in execution, execution lists are constructed for each method type. **Figure 10** depicts the sequence of steps followed by a single-tasking execution loop. Following initialization (step 2000), a time parameter is checked to see if the current time is less than the stop time (step 2010). If the time is not less than the stop time, the simulation ends (step 2020). If the time is less than the stop time, the simulation continues and the root output

method execution list is executed (step 2040). Following execution of the output method list

(step 2040) the update method execution list is executed (step 2060). Following the performance of an integrate step (2080)(the Integrate step is described more below in Figure 14), the time parameter is incremented by the applicable step size (step 2100).

Blocks are arranged in the single-tasking execution lists in the sorted order as shown in Figure 11A. A sorted list 2500 is used to generate an Output method execution list 2520 and an Update method execution list 2540. Referring back to the example in Figures 7 and 8, the engine sequentially steps through and execute each block in the block method execution list when the execution time divided by the sample time equals an integer number (1, 2, 3, 4, etc.). At time zero (T_0), all the blocks are executed. This involves executing the Output methods for blocks F, E, D, A, B, and C (in this order as dictated by the sorted list) and then executing the Update methods of blocks F, E, and D (again, in this order based on the sorted list). The execution time then is then incremented by step size, which in this case is assumed to be 0.1 seconds. Execution then commences once again at the top of the loop for $T=0.1$ ($T_{0.1}$). Blocks F and E have a sample time of 0.1 seconds and have a sample hit ($0.1 \div 0.1 = 1$, sample time is an integer multiple of the execution time), so the output block methods for Blocks F and E are executed. Block D, however, has a 1.0 second sample time and has no sample hit ($0.1 \div 1.0 = 0.1$, sample time is not an integer multiple of the execution time), so its output block method is not executed (essentially it is skipped). Block A, like Blocks F and E, has a 0.1 second sample time and so its output block method is executed. Blocks B and C, like Block D, have 1.0 second sample times and are skipped during this iteration of the simulation loop, which completes execution of the output block method execution list for $T_{0.1}$.

The execution timing of the example block diagram in single task mode is shown in the first time-line of Figure 11B. In this diagram, note that the execution-time is not synchronized with real-world time. Instead, execution time progresses as fast as it can in real-world time. The sorted list 2590 is executed on the time-line 2600. The methods in the list 2620 are executed at the appropriate time step 2640. Block diagram modeling software can also allow users to simulate real-world conditions by synchronizing execution time with real-world time. Such execution is illustrated in the second timing

diagram of Figure 11B. The methods 2620 are implemented at a time-step 2640 synchronized with real world time on the time line 2700.

In multitask mode, the engine performs execution along multiple time-lines based upon the number of block sample times used in the mode as shown in the flowchart of Figure 13. In the example of Figures 7 and 8, the model's blocks have a sample time of either 0.1 seconds or 1.0 second. This implies that the engine runs one set of blocks along a 0.1 second time line and another set of blocks along a 1.0 second time line. In order to run in multitask mode, the execution lists are first divided on the basis of methods (as in single-tasking mode) and then subdivided again based upon block sample times. This is illustrated in Figure 12A. The sorted list 2800 is used to generate an output method execution list 2820 and update method execution list 2880. The output method execution list 282 is split into two separate list execution lists 2840 and 2860 based on sample times. Similarly, the update method execution list 2880 is divided into two update method execution lists 2900 and 2920 based on sample times.

The execution engine uses the divided execution lists to create multiple execution time lines. In the multitask mode the engine places a higher execution priority on the faster sample time blocks than the slower sample time blocks. This prioritization is carried out by assigning Task Identification Numbers (TIDs) to each execution list; the higher the priority, the lower the TID. For example, a TID of 0 executes at a higher priority than a TID of 1, and so forth. Furthermore, because, during execution in multitask mode, execution transitions between the faster and slower blocks, and vice-versa, the multitask mode requires rate transition blocks that allow the model to transition from blocks running at fast sample times, in our example 0.1 seconds, to slower samples times, e.g., 1.0 seconds. The rate transition blocks are required to correctly simulate how a multi-rate system would behave in a real-time environment. To provide this transition, the engine promotes rate transition blocks to the TID of the fast block for which transition is being provided, although the engine executes these rate transition blocks at their slower rate. This is why Blocks D and B appear in the 0.1 sample time output method execution list in Figure 12A.

The execution of our example in the mult-itask mode may be seen in Figure 12B. At time $T=0$, the engine first executes the high priority output methods (those with TID 0) for Blocks F, E, D, A and B, then it executes the high priority update methods (those with TID 0) for Blocks F and E. After finishing the high priority blocks, the engine executes the lower priority output block methods (those with TID 1) for Block C, and then executes the lower priority update methods (those with TID 1), which, in this example, is Block D. In contrast to the single task mode, in multitask mode the engine runs through a TID inner loop to execute the output and update block methods before going on to the Integration step, as the flow chart in Figure 13 which is discussed below illustrates.

As a result of the inner TID loop, as well as the segregated block method execution lists, the order of execution in multitask mode differs from the order of execution in single task mode. Recall for the example that in single task mode that the order of execution at $T=0$ is: F_o , E_o , D_o , A_o , B_o , C_o , F_u , E_u , and D_u , where the subscript "o" stands for output method and the subscript "u" stands for update method. In the multitask mode, however, the order of execution at $T=0$ is: F_o , E_o , D_o , A_o , B_o , F_u , E_u , C_o , and D_u . Notice that C_o is executed in a different order in multitasking mode. This occurs because separate method execution lists (based upon sample time) are created and run in order from fastest sample time to slowest sample time. Additionally, the use of rate transition blocks restricts the connection of blocks with different rates. By requiring the insertion of these blocks into the model, the engine ensures that execution in multitask mode will follow the sorted list.

After it is finished executing the block methods for $T=0$, like in the single task mode, the execution time is incremented (again assume by 0.1 seconds) and execution goes to the beginning of the loop. The engine executes F_o , E_o , A_o , F_u , and E_u , and the engine does not execute the block methods of Blocks D, B, and C because the current execution time is not an integer multiple of those block's sample time. The engine repeats this execution until the execution time is incremented to 1.0 seconds, whereupon execution occurs in the same manner as described for $T=0$. The engine repeats this overall process until the execution stop time is reached.

Figure 12B shows two time-lines; the lower time-line 3060 represents the execution order of the faster sample time blocks (Blocks A, E, and F), along with the rate transition blocks (Blocks B and D), while the top time-line 3080 shows the execution order of the slower sample time block (Block C), and the rate transition (Block D) update method. The time-lines are generated from the sorted list 3020 and the associated sample times 3040. The lower line, representing the faster sample times has a TID of 0, and the top line has a TID of 1. For execution time $T=0$, the chart shows that the engine executes the output methods for Blocks F, E, D, A, and B (designated on the chart as F_o , E_o , D_o , A_o , B_o). Then, consistent with the flow chart for the multi-tasking mode (see Figure 13 discussed below), the engine executes the update block methods for Blocks F and E (designated F_u , and E_u). Once the engine is finished with the high priority block methods, the output method for Block C (C_o) and the update method for rate transition block D (D_u) are executed. The execution time is then incremented by the step size (continue to assume 0.1 seconds) and the blocks that have a sample hit are executed. The figure shows execution of F_o , E_o , A_o , F_u , and E_u , which is repeated, as noted above, until execution time equals 1.0 second. Notice, like in the non-real-time case for Single-task mode, the engine does not wait for time to elapse; rather it executes block methods immediately upon completion of the previous pass through the loop.

Figure 13 shows the overall sequence of steps taken by Simulink in multitask mode. Following initialization (step 2200), the output method execution list is executed for the fastest sample time (step 2220). The update method execution list is then executed for the fastest sample time (step 2240). A time parameter is checked (step 2250) to determine if the time is less than a designated stop time. If the stop time has been reached, the simulation completes (step 2260). Otherwise, the integrate stage (step 2280) is performed. The task ID variable is incremented (step 2300) and compared to a parameter of the number of sample times (step 2310). If the task ID is less than the number of sample times, the output method execution list for the methods assigned the new task Id are executed (2320) followed by the execution of the update method execution list assigned the new task ID (step 2340). The task ID variable is incremented and the process iterates with the task ID being compared to the number of sample rate times (step 2310). When the task ID number is determined to equal the number of sample rate times, the simulation time is incremented (step 2380) and the entire process

iterates with the output method list execution list (step 222) being executed for the fastest sample times. The process continues until the end of simulation when the time equals the stop time (step 2260).

5

In order to understand how the step size is picked within SimLoop, it is first necessary to understand the notion of a solver. The solver is a module of the execution engine that is responsible for performing two tasks: (a) determining how far execution time should be advanced between consecutive passes through the SimLoop in order to accurately trace the system's outputs, and (b) integrating the derivative of the states of the system to obtain the actual states. Based on how solvers perform the first task, they are generally classified into two basic classes: Fixed-step solvers or Variable-step solvers.

Fixed-step solvers are solvers in which the time step-size between consecutive passes through the SimLoop is a fixed quantity. The user generally explicitly specifies this quantity. These solvers are used to model types of systems that must operate within a defined time (discrete systems). For instance, an anti-lock braking system may be designed to control a car's braking system, and to execute such control in one-one hundredth (0.01) of a second so as to assure the car stops safely; if the braking system does not meet its timing constraints, the car may crash. Fixed-step solvers, therefore, are designed to help model discrete systems that have to generate a result in a fixed time period, and the fixed-step execution assures that the modeled system can generate such results.

Variable-step solvers are designed to model continuous systems where non-evenly spaced time steps are needed to simulate all significant behavior. For example, one may want to simulate the path of a bouncing ball, where it bounces, how high it bounces, and where it stops. It is known, based on experience, that the ball's bounces will not be evenly spaced, and that the height of the bounces will diminish as a result of gravity, friction, and other forces. Variable-step solvers are used for these types of continuous systems and to determine what step size to use so that the behavior of the ball will be accurately modeled.

The two broad classes of solvers are further subdivided based on the integration task they perform. There are several algorithms for carrying out numerical integration. The particular choice of the integration algorithm gives rise to the subclasses of solvers.

5 The difference in the conceptual definition of Fixed- and Variable-step solvers leads to the functional difference in the context of the SimLoop. The major difference between the solvers arises in the Integrate step of the SimLoop which is depicted in Figure 14. During the Integrate step, the Variable-step solver executes the Output and Derivative block method lists for a number of iterations that varies based on the solver
10 subclass (i.e., the numerical integration algorithm it uses) and integration error tolerances. In a fixed-step solver, the number of iterations is fixed for a given solver subclass. Another difference between solvers arises in the Integrate phase in the context of an operation known as zero-crossing detection. Zero-crossings in the derivatives of the state generally indicate a discontinuity in the states themselves. Because discontinuities often
15 indicate a significant change in a dynamic system, it is important to trace the system outputs precisely at such points. Otherwise, the outputs of the model could lead to false conclusions about the behavior of the system under investigation. Consider, again the example of the bouncing ball. If the point at which the ball hits the floor occurs between simulation steps, the simulated ball appears to reverse position in midair. This might lead
20 an investigator to false conclusions about the physics of the bouncing ball. To avoid such misleading conclusions, it is important that the execution has time steps on and around the vicinity of discontinuities.

 In the case of Fixed-step solvers, there is no notion of zero-crossing detection and
25 one is not guaranteed to find all points of discontinuity. One can only keep reducing the step-size to increase the probability of hitting the discontinuity. Contrastingly, in the case of Variable-step solvers, the Integrate step explicitly includes zero-crossing detection. The execution step size is then adjusted accordingly to ensure that discontinuities are tracked accurately. To enable zero-crossing detection, blocks that can produce
30 discontinuities instantiate a special execution method. This method registers a set of zero-crossing variables with the execution engine, each of which is a function of a state variable that can have a discontinuity. The zero-crossing function passes through zero from a positive or negative value when the corresponding discontinuity occurs. During

the zero-crossing detection phase of the Integration step, the engine asks each block that has registered zero-crossing variables to update the variables for the projected time of the next time step. These variables are then checked for a change of sign since the current step. Such a change indicates the presence of a discontinuity. An iterative process then
 5 tries to narrow down the location of the discontinuity and ensure that the next few time steps (at least 2) accurately bracket the location of the discontinuity. The final difference, which is in the step-size during execution, is a direct consequence of the two previous differences in the step-size determination. In Fixed-step solvers, the step size is a known and fixed quantity. For Variable-step solvers, the step size is determined during the
 10 integration iterations and the zero-crossing detection that happens during the Integration step.

An example of the variable-step solver is shown in Figure 14, the derivative method execution list is executed (step 2400) followed by the output method execution
 15 list (step 2420). The derivative method execution list is then executed again (step 2440) and the solver iterates between the execution of the output method execution list (step 2420) and the execution of the derivative method execution list (step 2440). A similar iteration loop then occurs between the execution of the output method execution list (step 2460) and the execution of the zero-crossing method execution list (step 2480). Note:
 20 that Simulink also includes other methods such as Projections and Jacobians in this step as needed.

While it is theoretically possible to have Variable-step solvers in the context of multitasking, such a combination is not employed in practice. This is because the step-
 25 size for such solvers can become very small making it impossible to keep up with the real-time constraint that generally goes along with multitasking execution. An added complication is that the integration step in such solvers is iterative and takes varying amounts of time at each step of the execution. Therefore, Variable-step solvers are generally used only in conjunction with the Single-Tasking SimLoop. Additionally, they
 30 are not usually employed in systems that need to operate in real-time.

When a model contains an algebraic loop, the engine calls a loop solving routine at each time step. The loop solver performs iterations and perturbations to determine the

solution to the algebraic condition (if it can). One possible approach to solving the algebraic equation $F(z) = 0$, is to use Newton's method with weak line search and rank-one updates to a Jacobian matrix of partial derivatives. Although the method is robust, it is possible to create loops for which the loop solver will not converge without a good
5 initial guess for the algebraic states z . Special blocks are generally provided to specify an initial guess of the states in the algebraic loop.

In addition to the various forms of the SimLoop, modeling packages such as Simulink use the output of the Link stage to compute linear models through a process
10 generally referred to as model linearization. These linear models may be used in the SimLoop at various points in the execution of the overall model. Alternatively, the linear model may be returned to the user. The linearization process involves the use of a Jacobian method defined on blocks and numerical Jacobian algorithm.

15 Information related to the compiled block diagram may be presented to users in an automatically generated report. This report allows users to quickly obtain documentation of the functional description of their model. Information related to the execution of a particular model (such as the time taken to execute various portions of the model and the coverage of various portions of the model) may be obtained automatically and presented
20 to the user as a report.

During automatic code generation, software source code is automatically produced from a graphical model. The software source code produced by the automatic code generation process may be compiled and then executed on a digital computer or
25 other electronic device implementing the functionality specified by the model. Current techniques for generating embedded code from graphical models are based on manual entry of embedded target characteristics using text files or simple dialog boxes. However, model developers often do not know how these target characteristics affect the generated code. The user is also required to enter parameters one-by-one to prepare for
30 generation of code, which is inefficient and time-consuming. The code generator is also not informed of what deployment goals or characteristics the model developer desires for the embedded software.

Summary of the Invention

In one embodiment, the present invention provides a tool for guiding a model developer through the process of preparing a model for embedded code generation. The software tool prompts the operator to input goals related to the generated code, and
5 generates feedback to the user regarding the compliance of the model with the specified goals. The software tool for guiding a model developer through the process of preparing a model for embedded code generation may be implemented using any suitable graphical based or text based program, such as a wizard. For example, the software tool may be an interactive wizard that implements hyperlinks to generate optimized target software for a
10 graphical model.

The software tool provides general classifications for many of the optimizations that yield the vast majority of software efficiency gains for RAM, ROM, and execution speed. High Level general characteristics could be obtained (e.g., choose between Fixed
15 point vs. Floating point code?). Once the general target details are known, the software tool can target specific details as needed to develop the embedded code. For example, the software tool can refine the general characteristics by user-supplied information (e.g., if Floating point choose ANSI-C or ISO-C math libraries). RAM, ROM, Stack space, clock frequency and many other hardware characteristics would also be obtained and refined
20 through this interactive wizard.

An accompaniment to the process is a model advisor that allows the user to select checks and analysis that should be done on the model prior to code generation. For example, one check would make sure that the model contains blocks suitable for code
25 generation. Another check would focus on ensuring that parameters within the blocks were set to yield optimized code. Search and replace capabilities can also be provided to facilitate substitution or modification of data within the model to yield better embedded software.

Brief Summary of the Drawings

Figure 1A depicts a dynamic system described with ordinary differential equations (ODE);

5 **Figure 1B** depicts a dynamic system described with difference equations;

Figure 1C depicts a dynamic system described with algebraic equations;

Figure 2 depicts components of a basic block diagram;

Figure 3 depicts the desired behavior of an integrator block;

10 **Figure 4** is a flow chart of the sequence of steps used to perform simulation of the block diagram;

Figure 5 depicts the replacement of a collection of blocks in a block diagram with an accumulator block;

Figure 6A depicts a block diagram and its associated directed graph;

15 **Figure 6B** depicts a linear sorted list generated from the directed graph of **Figure 6A**;

Figure 7A depicts an abstract example of a block diagram being executed;

Figure 7B depicts an abstract view of the execution methods instantiated by the blocks depicted in **Figure 7A**;

20 **Figure 7C** depicts a sorted list generated from the data dependencies between blocks of **Figure 7A**;

Figure 8 depicts a multi-rate system;

Figure 9 depicts the block diagram of **Figure 7A** and **Figure 8** with associated methods added to the blocks;

25 **Figure 10** is a flowchart of the sequence of steps followed by a single-tasking execution loop;

Figure 11A depicts the creation of execution lists from sorted lists in single task mode;

Figure 11B depicts the execution timing of block diagrams in single task mode in timelines synchronized and non-synchronized with real world time;

30 **Figure 12A** depicts the creation of execution lists from sorted lists in multi-task mode;

Figure 12B depicts the execution timing of block diagrams in multi-task mode;

Figure 13 is a flowchart of the overall sequence of steps taken by Simulink in multi-task mode;

Figure 14 is a flowchart of the sequence of steps followed by a variable-step solver.

5 **Figure 15** is a flow chart illustrating the steps involved in preparing a model for embedded code generation using a wizard according to an illustrative embodiment of the invention.

Figure 16A illustrates a graphical user interface displaying a block diagram suitable for implementation in the method shown in Figure 15.

10 **Figure 16B** is a graphical user interface showing instructions to the user upon initialization of the wizard from the block diagram of Figure 16A.

Figure 16C illustrates an interface for initiating the method shown in Figure 15.

Figure 17 illustrate a general code generation goals pane displayed by a wizard for prompting a user to enter general code generation goals according to an illustrative
15 embodiment of the invention.

Figures 18A and 18B illustrate the general code generation goals pane of Figure 17 after configuration of the parameters of the model in accordance with specified general goals.

Figures 19A-19E illustrate a model advisor pane displayed by a wizard according
20 to an illustrative embodiment of the invention.

Figure 20 illustrates the block diagram of Figure 16A during a model advisor process.

Figures 21A-21H illustrate a search and modify pane displayed by a wizard according to an illustrative embodiment of the invention.

25 **Figures 22A-22G** illustrate a detailed code generation goals pane displayed by a wizard according to an illustrative embodiment of the invention.

Figure 23 illustrates the detailed code generation goals pane after configuration of the parameters of the model in accordance with the specified detailed goals.

30 Detailed Description of the Invention

In an illustrative embodiment, the present invention provides a software tool in a graphical environment for generating optimized target software from a graphical model. In an illustrative embodiment, the software tool is an interactive wizard with hyperlinks

that guides model developers through the process of preparing a model for embedded code generation. However, one skilled in the art will recognize that a software tool for guiding a user through a process of preparing a graphical model for embedded code generation may be implemented using any suitable graphical based or text based program and that the invention is not limited to a wizard software program. For example, the invention can also be implemented using a parameter dialog box. The software tool prompts the user to input goals related to the generated code, and generates feedback to the user regarding the compliance of the model with the specified goals.

Wizards are software utilities used by applications to help users perform a particular task. Wizards help users perform complex tasks by guiding the user in a step by step fashion. A wizard may prompt a user for information, typically through a graphical user interface, in response to which the user supplies information to the wizard. The wizard may then employ this information, known as options or settings, to generate feedback to the user. For example, wizards have been added to software applications to help users configure peripherals such as modems and printers. Such a wizard guides the user through each step, such as selecting installation options, selecting ports, installing necessary software driver, creating necessary links to other applications, and setting any other necessary parameters.

Typically, the wizard attempts to guide the user as much as possible. For example, instead of forcing the user to type in a cryptic and hard to remember parameter, the wizard can provide a list of potentially acceptable parameters. As another example, the wizard may only provide as options those parameters that do not conflict with other goals, applications and/or devices. In such a way, the wizard guides the user through the otherwise difficult process.

Figure 15 is a flow chart illustrating the steps involved in preparing a model, illustrated as a block diagram, for embedded code generation to produce optimized target software using a wizard, according to an illustrative embodiment of the invention. The illustrated method can also be implemented using any suitable graphical or text-based program, and is not limited to a wizard. After building or modifying a block diagram or selecting a pre-constructed block diagram, such as the block diagram shown in **Figure**

16A, the user selects an appropriate system target file for the block diagram that is to be executed in step 110. In step 120, a user initiates the wizard, or other suitable software tool, which assists the user in preparing the model for embedded code generation. The user can launch the wizard from a “help” menu in the graphical modeling environment, by entering a text command or other command associated with the wizard, as shown in **Figure 16C**. In one embodiment, the wizard can display a graphical user interface (GUI) 410 explaining the steps involved in using the wizard, as shown in **Figure 16B**. As shown, the GUI 410 is launched from a button 409 in the GUI displaying the model 400.

After initialization, the wizard, or other suitable software tool, guides the user through the process of preparing the model for embedded code generation. The wizard first displays a user interface allowing the user to specify target specifications for the embedded code to be generated. In the preferred embodiment, when the graphical model preparation wizard is invoked, the wizard displays on the screen a user interface including a pane, which prompts the user to enter general configuration goals. In step 130, the user, in response to a query by the wizard, for example in the displayed pane, the user specifies general code generation goals for the model. The wizard thus allows a user to develop general classifications for many of the optimizations that yield the vast majority of software efficiency gains for RAM, ROM, and execution speed. High Level general characteristics can be obtained, for example, the user can choose between Fixed point vs. Floating point code, and specify other general goals. The inputs from the user on the user interface are processed programmatically to compare the selected goals of the user with the selected parameters of the model. After the user specifies the general code generation goals, in step 140, the wizard configures the parameters within the model based on the specified general code generation goals.

In step 150, the wizard may then allow the user to select checks and analysis that should be done on the model prior to code generation to ensure that the model is suitable for generating code and complying with the goals of the user. For example, one check would make sure that the model contains blocks suitable for code generation. Another check would focus on ensuring that parameters within the blocks are set to yield optimized code. After a user selects certain checks and analyses to be performed, the wizard programmatically processes the user selections, and based on the selections,

performs a check on the model. For example, the wizard can compare the objects of the model to a table that describes data types that are supported by certain blocks and identifies blocks that are suitable for production code generation. In the “model advisor” process of step 150, the wizard provides a report to the user on objects within the model that have parameters that are inconsistent with the general code generation goals specified in step 130.

In an optional step 160, the wizard searches for the objects in the current system that are identified in step 150 as being inconsistent with the general code generation goals and modifies the parameters of the inconsistent objects to comply with the general code generation goals. In one embodiment, the user can choose to manually modify the inconsistent objects. Alternatively, the system can automatically modify the inconsistent objects to comply with the goals. The search and modify capability facilitate substitution or modification of data within the model to yield better embedded software. After modifying the inconsistent objects, the wizard can return to the model advisor process in step 150 to check the model to verify that the updated parameters are consistent with the specified general code generation goals.

After ensuring that the updated parameters of the model are consistent with the general code generation goals specified in step 130, the wizard collects target specific details as needed to develop to embedded code in 170. The user, in response to a query by the wizard specifies detailed code generation goals through a user interface, such as the user interface used to enter the general goals in step 130 or a different user interface. In step 180, the user refines the general goals on a more detailed level. For example, if, in step 130, the user specified the code to be Floating point, in step 180, the user can select between ANSI-C or ISO-C math libraries. RAM, ROM, Stack space, clock frequency and many other hardware characteristics can also be obtained and refined through the interactive wizard. The wizard then processes the selections of the user and configures the model parameters based on the specified detailed code generation goals in step 180.

After execution of steps 110 through 180, the model parameters comply with the goals of the user. As a result, the user can generate code from the model in step 190 that

is consistent with the specified goals for the target characteristics, and deploy the generated code on the target.

FIGS. 17-23 are examples of windows, which show the process of preparing a model for code generation using a wizard or other suitable software tool. These windows are realized by a graphical user interface (GUI) 1000, and a mouse or a keyboard can be used as an input device, for example.

Figure 17 illustrate a graphical user interface 1000 displaying a general code generation goals pane 200 displayed by the wizard in response to initiation of the wizard by the user. The general code generation goals pane 200 prompts a user to execute step 130 by providing fields in which the user can specify the general code generation goals of a model according to an illustrative embodiment of the invention. As shown, the general code generation goals pane 200 displays a number of questions to the user related to general target characteristics for the code to be generated for the associated model 400. The user enters an answer to each question in an associated field to specify a code generation goal for the selected model. The wizard processes the answers of the user and, uses the specified code generation goals to prepare the model for embedded code generation.

For example, in the illustrative embodiment, the user can specify whether the code is Fixed point, Floating point or Mixed in field 210. In field 212, the user specifies whether or not the code generation options are specified for maximum efficiency. In field 214, the user assigns a priority to aspects of memory by specifying whether RAM or ROM is more important. In field 216, the user specifies how data is to be represented in code, by specifying whether the auto-generated identifiers are verbose or not. In field 218, the user specifies whether to include comments in the generated code, a primarily cosmetic feature affecting traceability. In field 220, the user specifies how to interface the generated code, i.e., whether to pass data as arguments, and so on. The general code generation goals pane 200 also includes a field 222 where the user can specify whether multiple models are to be combined into the same execution. In field 224, the user specifies whether the program should include an HTML report with the generated code.

After the user specifies one or more of the general code generation goals, the system configures the model properties based on the selected general goals, in step 140.

The user can instruct the system to configure the model properties by selecting the
5 “configure model” button 230 in the general code generation goals pane to initiate step 140. In step 140, the wizard programmatically processes the user selections in fields 210-224 and configures the model in accordance with the user selections. For example, in response to a user selection in field 210 for the target application to be “Fixed Point”, the wizard sets the code to only use integer code. Each user selection is processed by the
10 wizard, which updates the model, if necessary, based on the selections.

The user can specify any suitable number of general code generation goals in the pane 200. For example, the user can only choose to specify a few of the goals by answering only selected questions, while leaving the rest of the questions displayed in the
15 pane 200 unanswered. An unanswered question will set the associated goal to a default.

In the illustrative embodiment, the each of the fields 210-224 for specifying each goal includes a list of options, and the user selects a relevant goal from the list of options. However, one skilled in the art will recognize that any suitable mechanism for specifying
20 a goal related to the code to be generated for a graphical model may be used. For example, the field may comprise a blank space, in which the user may type a prescribed goal.

As shown in Figures 18A-18B, after configuring the model based on the general
25 configurations selected in step 130, the wizard updates the model 400 in step 140, and present the user with a list of the parameters of each object that have changed in response to the specified general code generation goals and the parameters of each object that have not changed. If desired, the user can undo the update by selecting the “undo button” 240, which cancels any changes made to the model. Otherwise, the user can click on the
30 hyperlink 250 to initiate the model advisor process of step 150. Alternatively, the user can click on the tab of the “model advisor” pane 320 to initiate step 150.

In step 150, the model advisor pane 320 prompts the user to select checks and analysis to be performed on the model prior to code generation. As shown in Figures 19A-19E, the model advisor pane 320 lists a number of conditions that can be checked based on the general code generation goals specified in step 130. Each condition includes a box, which the user can activate to instruct the wizard to check the associated condition for compliance with the specified goals, and to ensure that the model is connected and set up properly. The illustrative wizard also includes a “select all” button 322 to allow the user to simultaneously select all of the listed conditions, and a “deselect” button 324 to clear any previously made selections. In the illustrative embodiment, the conditions that may be checked by the wizard include, but are not limited to: identification of any questionable blocks within the specified system, identification of unconnected lines, input ports and output ports, validation of the system input specification, validation of the solver for code generation, identification of blocks that will generate expensive saturation and rounding code, checking of the system target file and validation of the target environment, identification of costly data initialization, identification of questionable interface specifications, identification of questionable code instrumentation, and so on.

After selecting the conditions to be checked, the user selects the “check model” button 326 to instruct the wizard to generate a report indicating whether the selected conditions are inconsistent with the specified goals. Figures 19B-19E illustrates an example of a report generated for the model 400 according to an illustrative embodiment of the invention. Figures 19B-19E show different, sequential sections of the same pane 320 after generation of the report, where each section is accessed using the scroll bar 1700 on the left side of the screen to scroll through the pane 200. As shown, the pane 320, after generation of the report, includes an explanation field after each selected condition for explaining whether the specified condition complies with the selected general code generation goals. If the specified condition does not comply, the wizard identifies the objects and parameters that are inconsistent with the specified condition. In each explanation field, the wizard uses hyperlinks to identify portions of the model that do not satisfy the selected condition. The hyperlinks link the report with an embodiment of the model, which highlights the inconsistent portion of the model associated with the selected hyperlink.

For example, as shown in Figure 19B, after the user requests the wizard to identify any questionable blocks in the model, the wizard processes the user request and, in field 341 below the questionable blocks box 360, identifies blocks in the model having settings that are not suitable for code generation. The wizard identifies questionable blocks by first creating a list of blocks in the model, then looping through the list of blocks and, for each block, checking a property that indicates whether the block is enabled for use with a code generation tool, such as Real-Time Workshop. The wizard then reviews predefined criteria for selecting blocks that are suitable for production code, consisting of a set of predefined rules and caveats, and compares the current settings of each block to the predefined criteria to determine whether the block is suitable for production code. For example, in the illustrative embodiment, the rules for selecting the blocks identify specify that a block is unsuitable for code generation when the generated code depends on absolute time, references non-finite values and/or contains instrumentation code that is only suitable for rapid prototyping. In addition, each block may only be suitable for code generation if the block has certain properties. For example, a block cannot depend on absolute time when inside a triggered subsystem.

In the illustrative embodiment, the wizard identifies the “first-order hold” block, the “rate limiter” block, and the discrete time integrator block, as being questionable blocks within the specified system. The wizard also includes an explanation as to why the identified blocks are questionable. As shown, hyperlinks 352, 354, 356 identify the inconsistent blocks. The hyperlinks link the report to the block diagram model so that, by clicking on each hyperlink, an interface displays the model with the relevant block highlighted. For example, in Figure 20, the “first order hold” block 352a is highlighted after clicking on the first-order hold hyperlink 352 in the report.

In box 370, the user requests the wizard to identify unconnected lines, input ports and output ports. In the field 371 below box 370, the wizard identifies unconnected lines, input ports and output ports of the model that are likely to cause problems propagating signal attributes. The wizard identifies the Look-Up Table 372 of the model 400 using a hyperlink 374, which, upon selection by the user, displays the model 400 with the Look-Up Table 372 to clearly identify the problem.

As shown in Figures 19C, by selecting box 380, the user can request the program to validate system inport specification. If box 380 is selected, the model advisor pane lists, in field 381 below box 380, the system inports that are not fully defined. Each system inport that is not fully defined is listed as a hyperlink 382a-382h. Each hyperlink
5 links the report to the associated inport of the model 400. To determine whether a block component, such as a system inport, is not fully defined, the wizard compares the block's settings to a series of predefined rules and caveats to determine suitability for code generation.

10 Figure 19C also shows the box 385 and associated field 386 for validating the solver for code generation. In the illustrative embodiment, the wizard informs the user that the user should change the solver to a fixed-step discrete solver before invoking the Real-Time Workshop Embedded Coder.

15 As shown in Figure 19C and the top of Figure 19D, box 390 allows the user to request the wizard to identify blocks that generate expensive saturation and rounding code. In field 391, the wizard identifies the blocks of the model that generate expensive saturation and rounding code using hyperlinks 392, 393 and 394. Each hyperlink links the report to the associated block of the model.

20 Figure 19D also illustrates the wizard output in response to a user's selection of box 410, where a user can request the wizard to check the system target file and validate the target environment. In field 411, which extends to Figure 19E, the wizard compares the final intended production target with the currently selected target. If the
25 configurations do not match, the Real-Time Workshop generates extra code to emulate the behavior of the production target. As shown, the wizard highlights elements of the configuration sets that are inconsistent with each other.

30 As shown in Figure 19E, which illustrates the bottom section of the report, the illustrative wizard indicates that the illustrative model 400 under consideration does not have any costly data initialization in field 413, below the box 414 in which the user can request the program to identify costly data initialization.

As also shown in Figure 19E, the wizard identifies questionable Simulink and Stateflow interface specifications in field 422, below box 421 in which the user can request the identification of questionable Simulink and Stateflow interface specifications. For example, field 422 in Figure 19E informs the user that the wizard has detected a weak data type specification for the Simulink I/O, and advises the user to update the chart identified in hyperlink 424 to use strong data typing for the I/O.

Below field 422, the wizard indicates that the model does not have any questionable code instrumentation in field 433, in response to a user request specified in box 434 to identify questionable code instrumentation.

Box 451, which allows the user to request the program to validate sample times and tasking mode, and box 453, which allows the user to request the program to identify questionable fixed-point operations remain unchecked, and the wizard informs the user that these conditions cannot be checked if the model is not compiled.

One skilled in the art will recognize that the model advisor pane 320 shown in Figure 19A-19E is illustrative of one embodiment of the invention, and that any suitable means for providing a report to the user regarding compliance of parameters of a model with selected conditions can be used.

After reviewing the report in pane 320, the user can select the “search and modify” pane 500, to initiate step 160, which allows the user to search for one or more of the characteristics of the model identified in the report and implement one or more of the changes advised by the wizard to enhance the embedded software produced from the model. Figures 21A-21D illustrate different sections of the search and modify pane 500 during execution of step 160. The user can choose which objects and which parameters to modify, or allow the program to automatically tweak the inconsistent parameters to comply with the specified goals.

As shown, the search and modify pane 500 implements a plurality of hyperlinks for performing the modification step 160. For example, the “Simulink object search” hyperlink 511 scrolls the pane 500 to a section 515 of the pane where the user can specify

a general Simulink object search and modify action. In field 512, the user specifies that the Simulink object to be modified is a block. In field 513, the user enters search criteria, for example, relating to the block type, block name and block description. After specifying the search criteria, the user selects the “search” button 516 to initiate the search and replace process. As shown in Figure 21C, the program updates the attributes that satisfy the search conditions specified by the user and displays the update to the user. In the illustrative embodiment, the program updates a gain1 block and a gain2 block, and includes hyperlinks linking the search and modify pane 500 to the particular block that has been changed. If desired, the user can cancel the changes by selecting the “undo update block 502.

A “continue” hyperlink 507 shown in Figure 21B causes the wizard to display the “frequent tasks” section 522 of the search and modify pane 500, as shown in Figure 21C. The “frequent tasks” section 522 can also be accessed through the “frequent tasks” hyperlink 521 shown in Figure 21A. In the frequent tasks section 522, the user instructs the program to perform selected tasks. For example, by selecting box 534, the user can request the program to saturate on integer overflow and specify a new value. After selecting which tasks to be performed, the user selects the “modify” button 538 to initiate modification of all objects for the selected tasks.

After modifying the parameters in step 160 to comply with the general code generation goals, the user can return to step 150 to ensure that the modified parameters comply with the specified goals.

Alternatively, the user can initiate step 170 by selecting the “detailed code generation goals” pane 600, which is illustrated in Figures 22A-22I. If the model advisor indicates that no changes are necessary or advised, the user can initiate step 170 directly, without executing step 160. In step 160, the user optimizes the system and tweaks specific goals for the embedded code to be generated from the selected model at a lower level than the general goals specified in step 130. As shown, in the “detailed code generation goals” pane 600 the wizard collected target specific details as needed to develop the embedded code. For example, the top portion of illustrative pane 600, shown in Figure 22A, includes a list of hyperlinks, where each hyperlink relates to a general

configuration goal. By clicking on a hyperlink, a user is directed to a field for inputting specific goals for the embedded code. The specific goals associated with each hyperlink are a subset of the general goal indicated by the selected hyperlink and are more detailed than the general goal specified in the “general code generation goals” pane.

5

For example, by clicking on the “target environment” hyperlink 610, the interface 1000 scrolls down to the “target environment” field 612 of the pane 600, shown in Figure 22B, wherein a user can specify additional goals related to the target environment. In the exemplary embodiment, the target environment field 612 allows a user to specify certain
10 detailed goals, such as the tasking mode in field 613, whether to generate a concise example main program for the associated mode in field 614, the target operating system in field 615, whether to generate integer code only in field 616, the target floating point math environment in field 617, the production hardware characteristics in field 618 and the microprocessor word sizes in fields 619a-610d.

15

The “block diagram optimizations” hyperlink 620 brings the user to a “block diagrams optimizations” field 622 of the pane 600, shown in Figure 22C, wherein the user can specify additional goals related to the efficiency of the code generation. For example, the user can instruct the user to eliminate redundant blocks in field 624. The user can also
20 specify whether to conditionally execute blocks without state that feed switch operations in field 626.

The “data” hyperlink 630 brings the user to a “data” field 632 of the pane 600, shown in Figures 22C and 22D. In the “data” field 632, the user enters specific goals
25 related to initialization, use of bitsets and custom storage classes. For example, in field 634, the user instructs the program whether to initialize root level I/O data to zero, and, in field 635, whether to initialize internal state data to zero. In field 636, the user can specify whether to explicitly initialized floats and doubles to zero. In fields 637 and 638, the user can specify whether to use bitsets for storing state information in Stateflow charts
30 and/or whether to use bitsets for storing Boolean data in Stateflow charts, respectively. In field 639, the user can instruct the program to ignore custom storage classes.

The “parameters” hyperlink 640 at the top of the pane 500 links to a “parameters” field 642 of the pane 600, shown in Figures 22D and 22E. In the parameters field 642, the user can specify detailed parameters goals. For example, in field 644, the user sets inline parameter values on or off. In field 645, the user instructs the program how to generate scalar inlined parameters, and in field 646, the user instructs the user whether to optimize storage of non-scalar parameter values. In field 647, the user enters the parameter structure implementation.

The “signals” hyperlink 650 brings the user to a “signals” field 652, shown in Figures 22E and 22F, in which the user can specify detailed signals goals. For example, in field 651, the user specifies whether to implement every signal in persistent member, and whether to reuse local variables in field 653. In field 654, the user can specify whether to eliminate superfluous temporary variables. The user can specify whether to preserve integer downcasts in folded expressions and whether to expression fold unrolled vector signals in fields 655 and 656, respectively. In field 657, the user can instruct the program to implement logic signals as Boolean data. In field 658, the user specifies a threshold for a signal width and instructs the program to unroll for loops when the signal width does not exceed the specified threshold. In field 659, the user can specify whether to inline variable signals with macros.

The “traceability” hyperlink 660 brings the user to a traceability field 662, as shown in Figures 22F and 22G, in which the user can specify detailed goals relating to the traceability of the embedded code. For example, in field 663 the user can alter the goal of whether to include comments in the generated code, which was specified in the general code generation goals pane. In fields 663a-663c, which are subsets of field 663, the user specifies whether to insert Simulink block descriptions, whether to show eliminated code statements, and whether to unconditionally comment parameter structure data. The traceability field 662 also allows the user to specify certain goals relating to the identifiers. For example, in field 664, the user can specify the maximum identifier length. In field 665, the user can specify whether to prefix the model name to global identifiers. In field 668, the user can specify whether to include the system hierarchy number in the identifiers, and in field 667, the user can specify whether to include a data type acronym in the identifiers.

The “reporting” hyperlink 670 brings the user to a reporting field 672, shown in Figure 22H where the user can specify additional reporting goals. For example, in field 671, the user can instruct the program to show a build log inside a MATLAB command window. In field 673, the user can instruct the program to document generated code inside an HTML report.

The “validation” hyperlink 675 brings the user to a validation field 676, where the user can specify additional validation goals. For example, in field 677, the user can specify whether to create a Simulink block for software-in-the-loop testing. In field 678, the user can instruct the program to instrument the generated code to log results into a MAT-file.

The “model interfaces” hyperlink 680 brings the user to a model interfaces field 682, where the user can specify additional goals related to the interfaces for the model. For example, in field 683, the user can instruct the program to generate reusable code for the entire model. Fields 683a and 683b, which are subsets of field 683, allow the user to specify a reusable code error diagnostic and how to pass root level I/O data. In field 684, the user can instruct the program to suppress error status in real-time model data structure. In field 685, the user can instruct the program to combine the model step function into a single output/update function. In field 686, the user can specify whether to generate a model termination function. In field 687, the user can specify whether to generate an ASAP2 data exchange file. The user can also request a C-interface API for runtime signal monitoring in field 688a and/or for parameter tuning in field 688b. In field 689, the user can instruct the program to instrument code for a Simulink External Mode.

As shown, the fields for each general goal are in sequential order in the pane 600, according to the order of the hyperlinks, so that a user can scroll down the user interface to reach the sections related to each goal. Each field also includes a “back to top” hyperlinks to bring user to the top of the pane 600.

After specifying the detailed configuration goals, the user can apply the specified goals, for example, by selecting an “apply button” 690, as shown in Figure 22I. In

response to the user request to apply the specified goals, the program modifies certain parameters of the model that change in response to the detailed goals. As shown in Figure 23 the wizard displays a list to the user on the user interface 1000 describing the objects and parameters that have changed in response to the detailed goals. The program can also list the objects and parameters that remain unchanged. After reviewing the updated objects and parameters, the user can cancel the changes by selecting the “undo update” button 691. The user can accept the updates by selecting the “continue” button 692

After specifying the detailed goals and modifying any objects changed by any changes to the detailed goals, the model is ready to generate code and deploy the generated goal on the specified target.

One skilled in the art will recognize that any suitable mechanism for initiating each step may be implemented. For example, the wizard may automatically guide the user through each step, or the user may initiate a step by selecting an appropriate pane, button, hyperlink or other object displayed on a graphical user interface. The user can toggle between the different panes and steps.

The present invention has been described relative to an illustrative embodiment. Since certain changes may be made in the above constructions without departing from the scope of the invention, it is intended that all matter contained in the above description or shown in the accompanying drawings be interpreted as illustrative and not in a limiting sense.

It is also to be understood that the following claims are to cover all generic and specific features of the invention described herein, and all statements of the scope of the invention which, as a matter of language, might be said to fall therebetween.